Greg Surma Follow
https://gsurma.github.io
Sep 26 · 6 min read

Cartpole - Introduction to Reinforcement Learning (DQN - Deep Q-Learning)

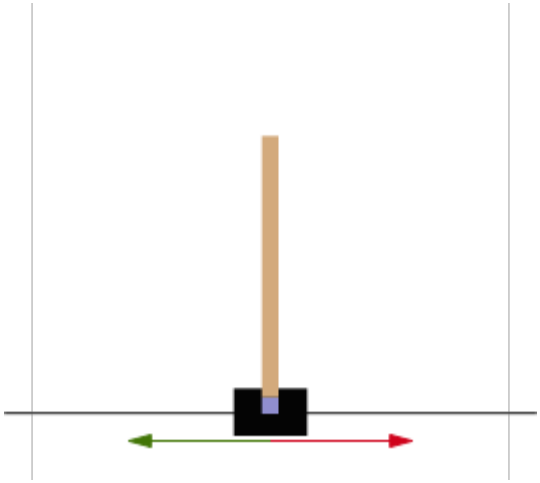
In today's article, I am going to introduce you to the hot topic of Reinforcement Learning. After this post, you will be able to create an agent that is capable of learning through trial and error and ultimately solving the cartpole problem.

Before and after training / Before and after reading this article

Table of Contents

Cartpole Problem

Cartpole - known also as an Inverted Pendulum is a pendulum with a center of gravity above its pivot point. It's unstable, but can be controlled by moving the pivot point under the center of mass. The goal is to keep the cartpole balanced by applying appropriate forces to a pivot point.



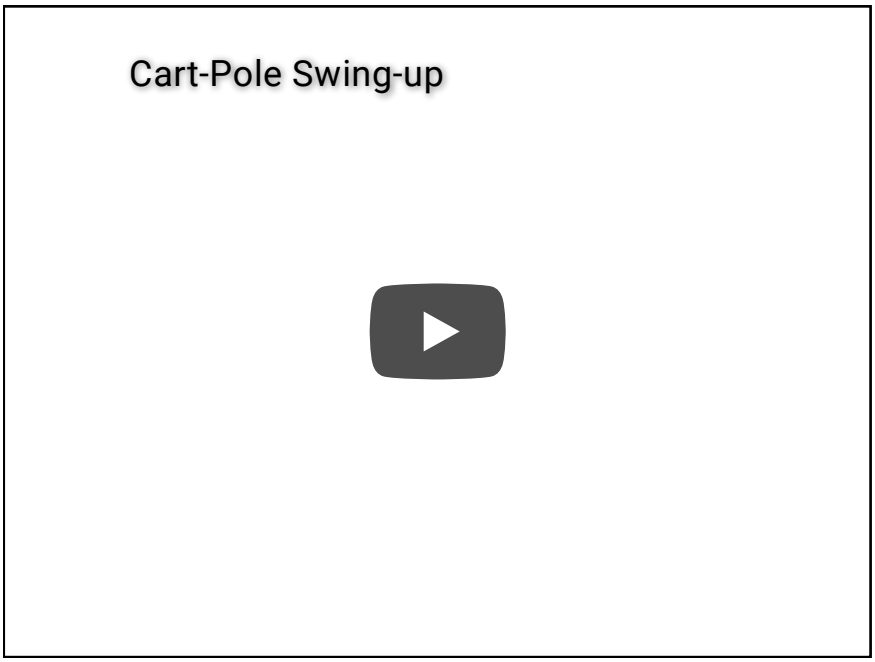
Cartpole schematic drawing

Violet square indicates a pivot point

Red and green arrows show possible horizontal forces that can be applied to a pivot point

> *A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart. The pendulum starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.*

Take a look at a video below with a real-life demonstration of a cartpole problem learning process.



Real-life application of Reinforcement Learning

Looks cool, right?

Implementing such a self-learning system is easier than you may think. Let's dive in!

Reinforcement Learning

In order to achieve the desired behavior of an agent that learns from its mistakes and improves its performance, we need to get more familiar with the concept of Reinforcement Learning (RL).

RL is a general concept that can be simply described with an agent that takes actions in an environment in order to maximize its cumulative reward. The underlying idea is very lifelike, where similarly to the humans in real life, agents in RL algorithms are incentivized with punishments for bad actions and rewards for good ones.

Let's move on from this high-level overview to an actual implementation in code.

Gym

Cartpole project is available here and as usual, I recommend reading the README file first.

gsurma/cartpole

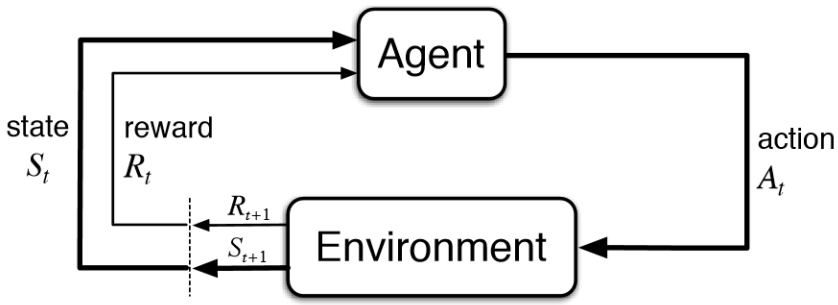
OpenAI's cartpole env solver.
Contribute to gsurma/cartpole
development by creating an account
on GitHub.

github.com

Project is based on top of OpenAI's gym and for those of you who are not familiar with the gym - I'll briefly explain it.

Long story short, gym is a collection of environments to develop and test RL algorithms. Cartpole is one of the available gyms, you can check the full list here. It's built on a Markov chain model that is illustrated below.

Markov Chain



Markov chain - SARSA'

We start with an initial environment. It doesn't have any associated reward yet, but it has a state (S_t).

Then for each iteration, an agent takes current state (S_t), picks best (based on model prediction) action (A_t) and executes it on an environment. Subsequently, environment returns a reward (R_t+1) for a given action, a new state (S_t+1) and an information if the new state is terminal. The process repeats until termination.

This may sound overwhelming at first, but you'll see over time how logical it is.

Let's see how it actually looks in code.

```
1   def cartpole():
2       env = gym.make("CartPole-v1")
3       observation_space = env.observation_space.shape[0
4       action_space = env.action_space.n
5       dqn_solver = DQNSolver(observation_space, action_
6       while True:
7           state = env.reset()
8           state = np.reshape(state, [1, observation_spa
9           while True:
10              env.render()
11              action = dqn_solver.act(state)
12              state_next, reward, terminal, info = env.
13              reward = reward if not terminal else -rew
```

Lines 2 - 5

Initially, we create our environment and an agent (DQNSolver) with an observation space (possible state values) and an action space (possible actions that can be performed), see below:

```
1       Observation:
2           Type: Box(4)
3           Num     Observation               Min
4           0       Cart Position             -4.8
5           1       Cart Velocity             -Inf
6           2       Pole Angle                 -24°
7           3       Pole Velocity At Tip      -Inf
8
9       Action:
```

Lines 7 - 8

For each run, we initialize new environment and set its initial state.

Lines 10 - 19

For each step until termination, based on a given state, we get an action from an agent. Then we execute it on an environment and receive a new state along with the associated reward. Afterward, we

remember our SARS' (state, action, reward, state_next, terminal) and perform Experience Replay.

Wait, what? Why would we need to remember our actions and do anything with them?

To find out why, let's proceed with the concept of Deep Q-Learning.

Deep Q-Learning (DQN)

DQN is a RL technique that is aimed at choosing the best action for given circumstances (observation). Each possible action for each possible observation has its Q value, where 'Q' stands for a quality of a given move.

But how do we end up with accurate Q values? That's where the deep neural networks and linear algebra come in.

For each state experienced by our agent, we are going to remember it

```
dqn_solver.remember(state, action, reward, state_next,
terminal)
```

and perform an experience replay.

```
dqn_solver.experience_replay()
```

Experience replay is a biologically inspired process that uniformly (to reduce correlation between subsequent actions) samples experiences from the memory and for each entry updates its Q value.

```
1   def experience_replay(self):
2       if len(self.memory) < BATCH_SIZE:
3           return
4       batch = random.sample(self.memory, BATCH_SIZE)
5       for state, action, reward, state_next, terminal i
6           q_update = reward
7           if not terminal:
8               q_update = (reward + GAMMA * np.amax(self
```

Line 8 is crucial here. We are calculating the new q by taking the maximum q for a given action (predicted value of a best next state), multiplying it by the discount factor (GAMMA) and ultimately adding it to the current state reward.

In other words, we are updating our Q value with the cumulative discounted future rewards.

Here is a formal notation:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

(source: https://en.wikipedia.org/wiki/Q-learning)

For those of you who wonder how such function can possibly converge, as it looks like it is trying to predict its own output (in some sense it is!), don't worry - it's possible and in our simple case it does.

However, convergence is not always that 'easy' and in more complex problems there comes a need of more advanced techniques that stabilize training. These techniques are for example Double DQN's or Dueling DQN's, but that's a topic for another article (stay tuned).

Deep Neural Net

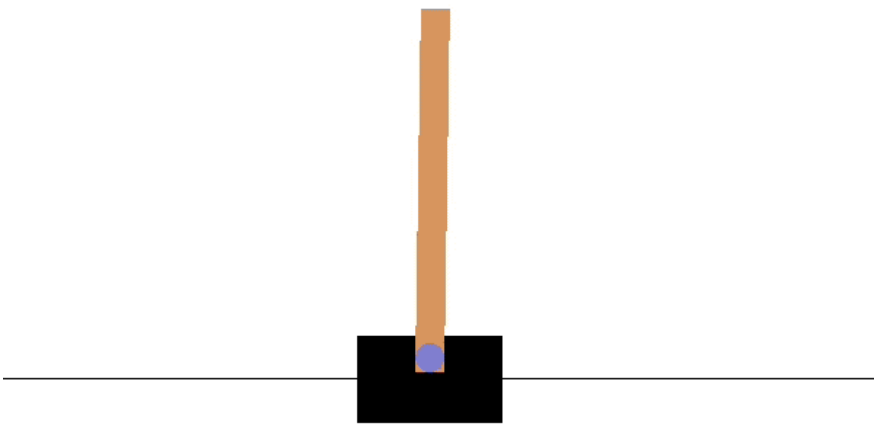
Lastly, let's take a look on our neural net architecture:

```
    self.model = Sequential()
    self.model.add(Dense(24, input_shape=(observation_spac
    self.model.add(Dense(24, activation="relu"))
    self.model.add(Dense(self.action_space, activation="li
```

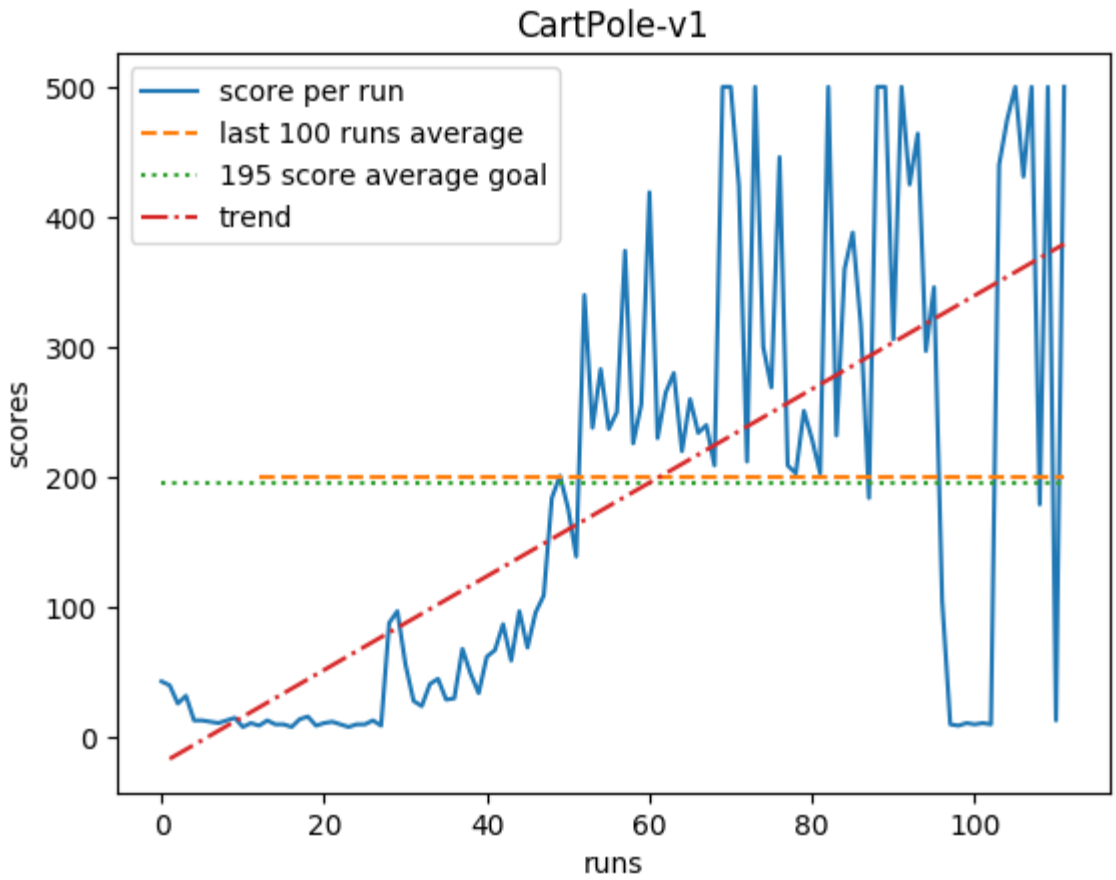
Its size is suited for the project's modest complexity, but feel free to play with its structure and parameters.

Learning Performance

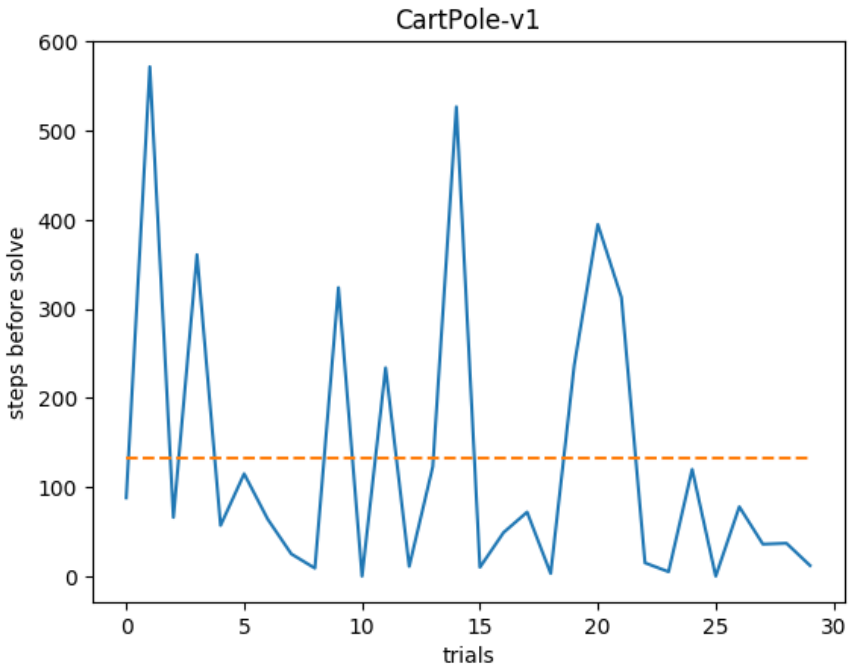
Finally, let's see how our agent performs.



Trained cartpole example



Trial overview example



Solved trials overview

> CartPole-v0 defines “solving” as getting an average reward of 195.0

> over 100 consecutive trials.

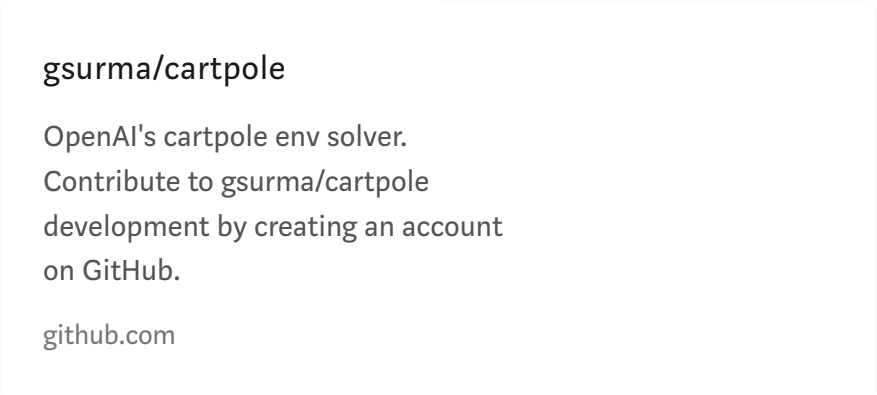
Our algorithm solves cartpole on average in ~131 'steps before solve'.

What's next?

I challenge you to create your own RL agents! Let me know how they perform in solving the cartpole problem. Furthermore, stay tuned for the new articles.

. . .

Don't forget to check the project's github page.



. . .

Questions? Comments? Feel free to leave your feedback in the comments section or contact me directly at https://gsurma.github.io.